Gemeinsam gegen den Ball-of-Mud

Peter Gafert

15.12.2022

About me

- Peter Gafert (@codecholeric)
- Principal Consultant@TNG Technology Consulting GmbH
- Everyday agile architect
- Creator of ArchUnit

Disclaimer: Der Vortrag fokussiert sich darauf, wie man Probleme mittels ArchUnit lösen kann. Es kann durchaus sein, dass es in konkreten Projekten bessere Wege gibt (Java-Modules?) oder gewisse Probleme schon durch das Build-Tool gelöst sind.

Überblick

Wir widmen uns einer mittelkomplexen Anwendung zur Verwaltung von IOT-Device-Produktion

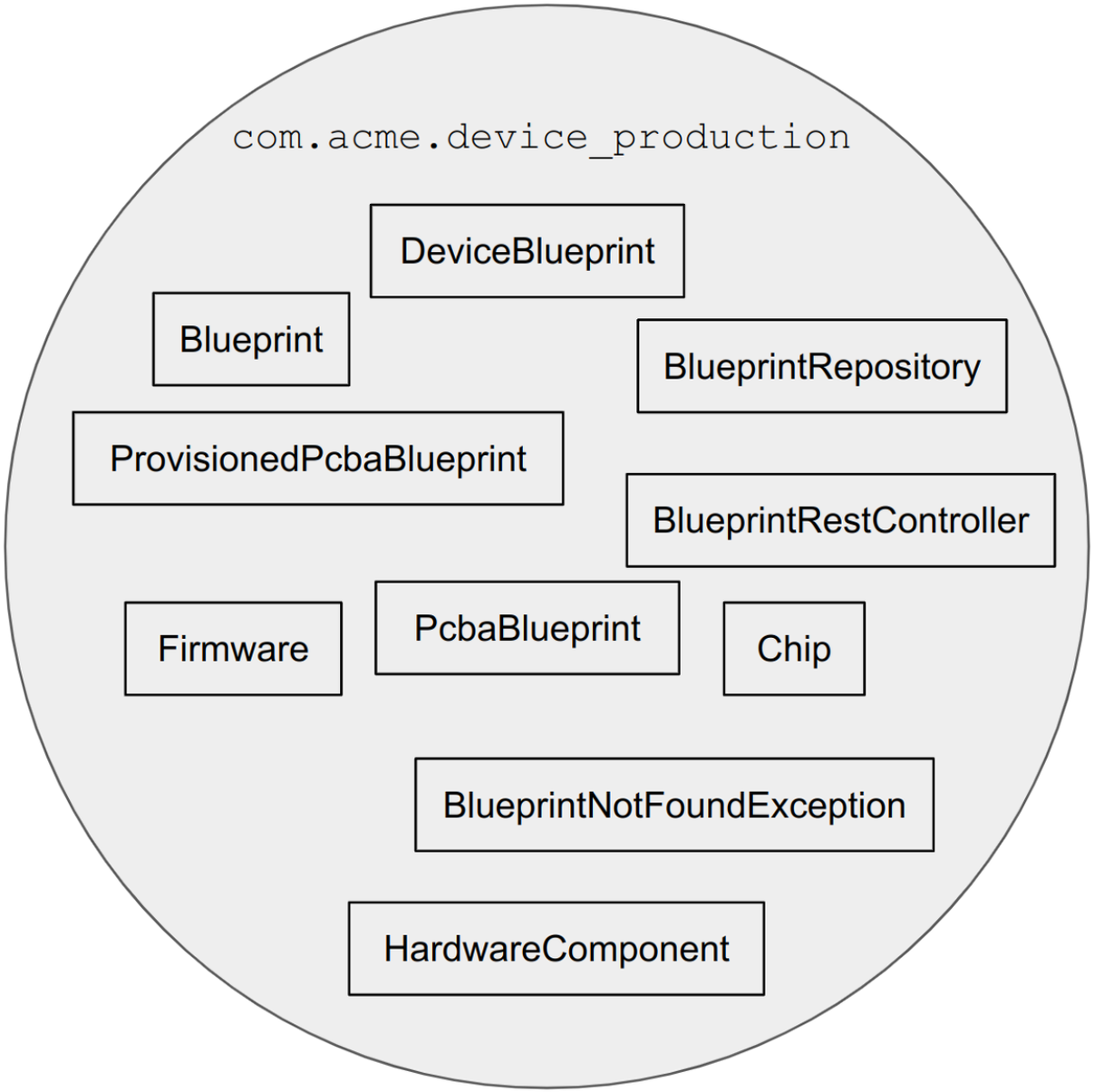
Vision

- Wir verwalten Blueprints auf deren Grundlagen verschiedene Devices erstellt werden können
- Die Produktion besteht aus mehreren Schritten: PCBA, Provisioned PCBA, Device, Product
- Wir wollen auch Packaging verwalten, d.h. Verpackung mehrerer Products in Masterboxes und Zuordnung auf Pallets
- Products sollen auch Business-Partnern zugeordnet werden können und dann spezielles Branding erhalten

Überblick

Und am Anfang stand...

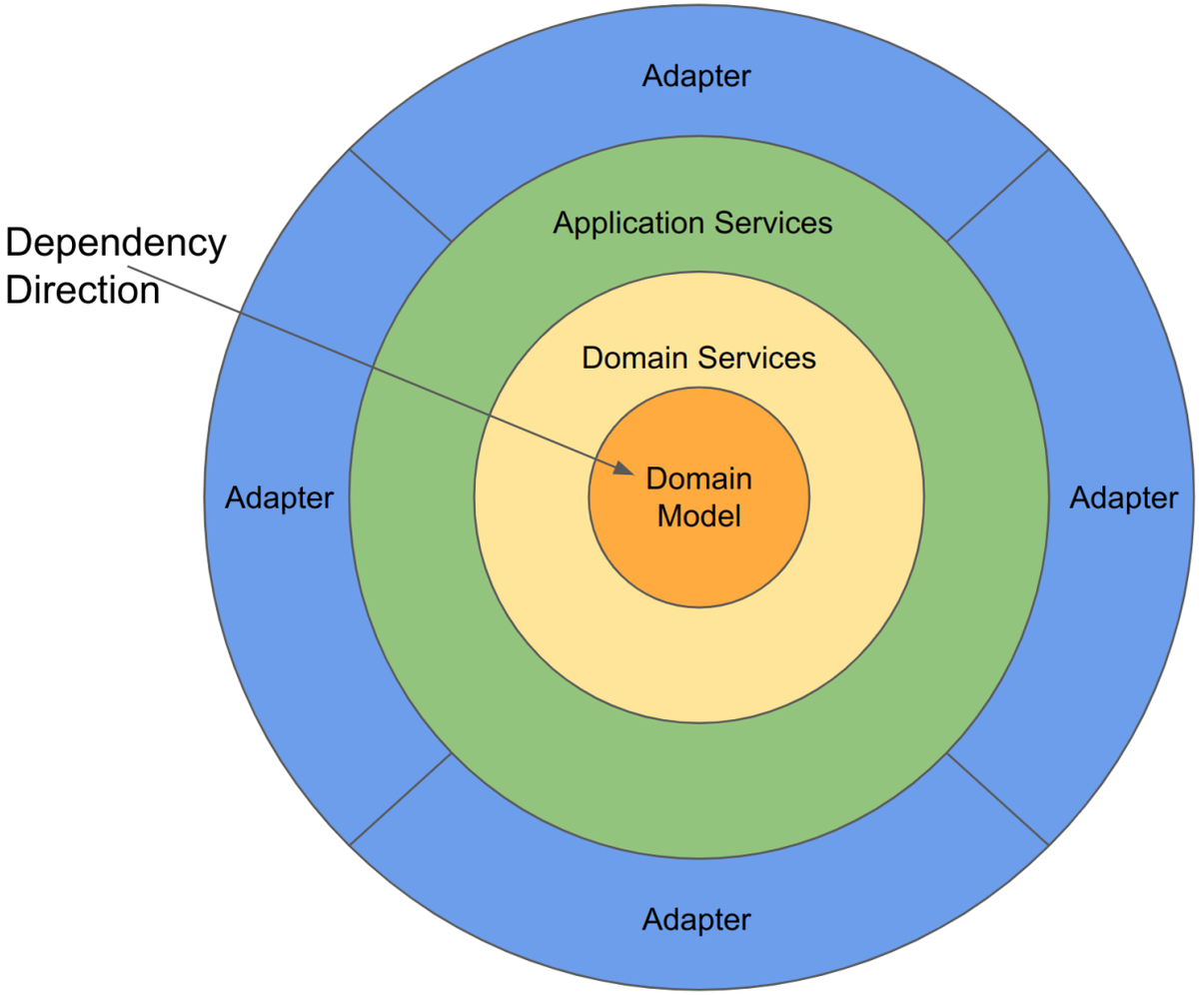
… der POC


com.acme.device_production
DeviceBlueprint
Blueprint
BlueprintRepository
ProvisionedPcbaBlueprint
BlueprintRestController
Firmware
PcbaBlueprint
Chip
BlueprintNotFoundException
HardwareComponent

Zeit für technische Struktur

- wir erwarten eine mittel-komplexe Domäne
- und entscheiden uns im Team für eine Onion-Architektur

Onion-Architektur



Team-Entscheidung festhalten

- z.B. als (A)rchitecture (D)ecision (R)ecord
- mittels ArchUnit als Test definieren

Onion-Architektur in ArchUnit

Wir könnten einzelne Regeln schreiben

```
@ArchTest
static final ArchRule domain_model_should_be_independent =
  classes().that().resideInAPackage("..domain.model..")
    .should()
    .onlyDependOnClassesThat().resideInAPackage("..domain.model..");

// many more rules to come
```

Oder LayeredArchitecture ”missbrauchen”

```
@ArchTest
static final ArchRule adheres_to_Onion_Architecture =
  layeredArchitecture()
    .consideringOnlyDependenciesInAnyPackage("com.acme.device_production..")

    .layer("Domain Model").definedBy("..domain.model..")
    .layer("Domain Service").definedBy("..domain.service..")
    .layer("REST UI Adapter").definedBy("..adapter.rest_ui..")
    .layer("REST Manufacturing Adapter").definedBy("..adapter.rest_manufacturing..")

    .whereLayer("Domain Model").mayNotAccessAnyLayer()
    .whereLayer("Domain Service").mayOnlyAccessLayers("Domain Model")
    .whereLayer("REST UI Adapter")
    .mayOnlyAccessLayers("Domain Model", "Domain Service")
    .whereLayer("REST Manufacturing Adapter")
    .mayOnlyAccessLayers("Domain Model", "Domain Service");
```

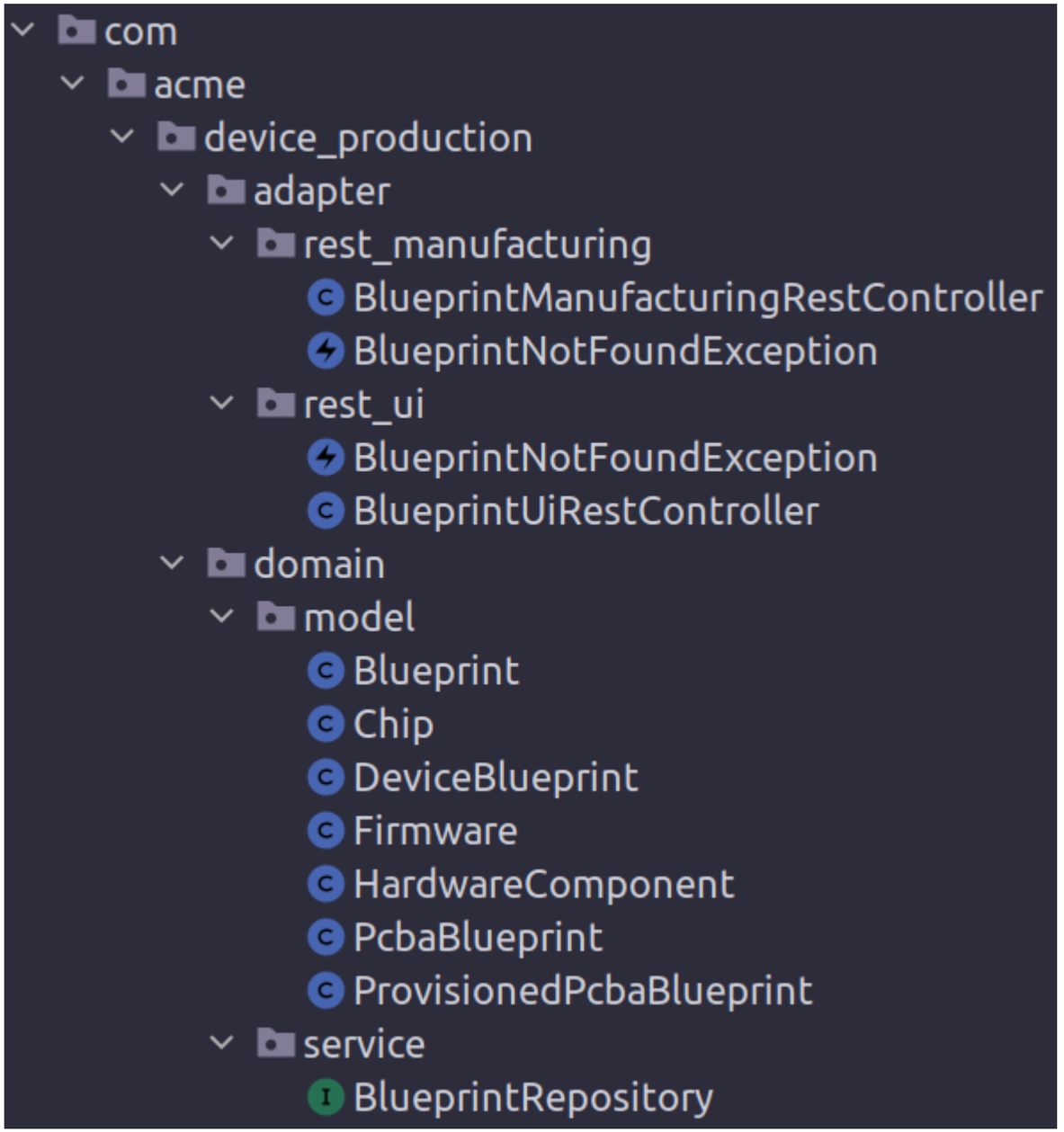
Oder falls passend: OnionArchitecture

```
@ArchTest
static final ArchRule adheres_to_Onion_Architecture =
  onionArchitecture()
    .domainModels("..domain.model..")
    .domainServices("..domain.service..")
    .adapter("REST UI", "..adapter.rest_ui..")
    .adapter("REST Manufacturing", "..adapter.rest_manufacturing..")
    .withOptionalLayers(true)
    .ensureAllClassesAreContainedInArchitecture();
```

Bei Verletzung

```
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
Rule 'Onion architecture consisting of (optional)
domain models ('..domain.model..')
domain services ('..domain.service..')
adapter 'REST UI' ('..adapter.rest_ui..')
adapter 'REST Manufacturing' ('..adapter.rest_manufacturing..')' was violated (6 times):
Method <c.a.d.adapter.rest_ui.BlueprintUiRestController.getDeviceBlueprint(long)>
  calls constructor <c.a.d.adapter.rest_manufacturing.BlueprintNotFoundException.<init>(long)>
  in (BlueprintUiRestController.java:41)
...
```

Nun sieht unsere Struktur so aus



Technische Konventionen

- Beispiel: Verwirrung über Validierung
- wir diskutieren im Team wo wir i.A. Validierung durchführen wollen
- wir entscheiden uns für ein Builder-Pattern im Domain-Kontext

Builder-Konvention in ArchUnit

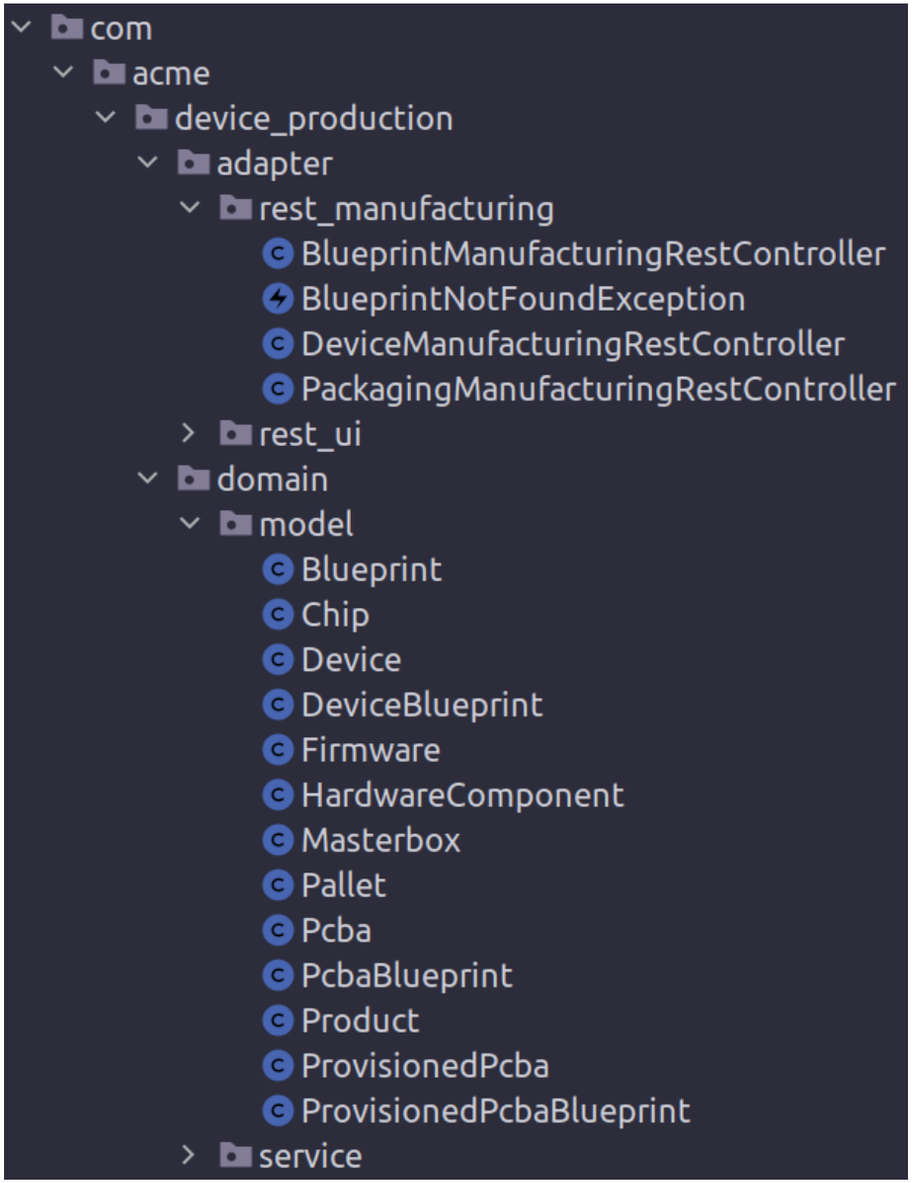
```
@ArchTest
static final ArchRule domain_objects_should_use_builder_pattern =
  classes()
    .that().resideInAPackage("..domain.model..")
    .and().haveNameNotMatching(".*\\$Builder")
    .should(new ArchCondition<>("follow the builder pattern") {
      @Override
      public void check(JavaClass clazz, ConditionEvents events) {
        clazz.getConstructors().stream()
          .flatMap(constructor -> constructor.getAccessesToSelf().stream())
          .filter(access -> !access.getOriginOwner().getClassHierarchy()
                               .contains(access.getTargetOwner()))
          .filter(access -> !access.getOriginOwner().getName()
                               .equals(access.getTargetOwner().getName() + "$Builder"))
          .forEach(access -> events.add(
                               SimpleConditionEvent.violated(clazz, access.getDescription())));
      }
    });
```

Bei Verletzung

```
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
Rule 'classes that reside in a package '..domain.model..'
and have name not matching '.*\$Builder'
should follow the builder pattern' was violated (1 times):
Constructor <c.a.d.p.d.m.Pallet.<init>()>
  calls constructor <c.a.d.p.d.m.Masterbox.<init>()>
  in (Pallet.java:5)
```

Überblick

Fehlen fachlicher Kohäsion



Diskussion der Domäne(n)

- Ergebnis: Wir haben nun 4 (Sub-)Domänen
- wir finden Namen
- und legen erlaubte Abhängigkeiten fest
- => Blueprint, Manufacturing, Packaging, Partner

Fachliche Module in ArchUnit

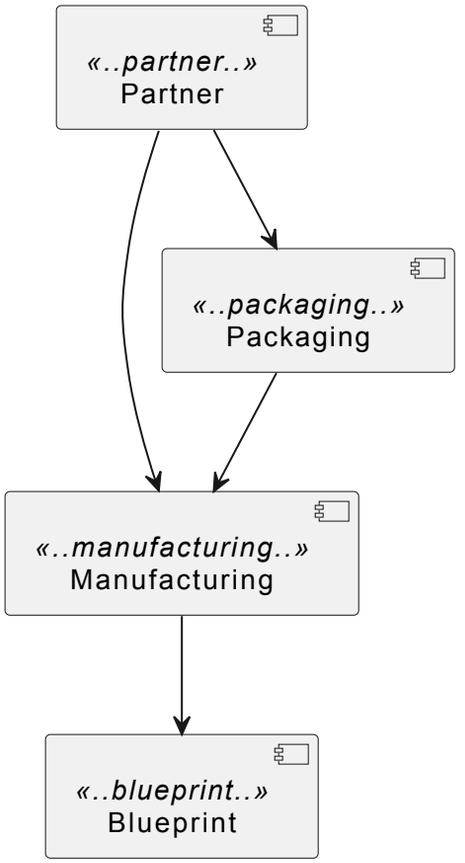
Z.B. via PlantUML Regel (1)

```
@startuml

[Blueprint] <<..blueprint..>>
[Manufacturing] <<..manufacturing..>>
[Packaging] <<..packaging..>>
[Partner] <<..partner..>>

[Manufacturing] --> [Blueprint]
[Packaging] --> [Manufacturing]
[Partner] --> [Manufacturing]
[Partner] --> [Packaging]

@enduml
```



Z.B. via PlantUML Regel (2)

```
@ArchTest
static final ArchRule should_respect_module_structure =
  classes().should(
    adhereToPlantUmlDiagram(module_diagram_url,
      consideringOnlyDependenciesInAnyPackage("com.acme.device_production.."))));
```

Oder via Slices-API (1)

```
static final Map<String, Set<String>> allowedModuleDependencies = Map.of(
  "manufacturing", Set.of("blueprint"),
  "packaging", Set.of("manufacturing"),
  "partner", Set.of("manufacturing", "packaging")
);
```

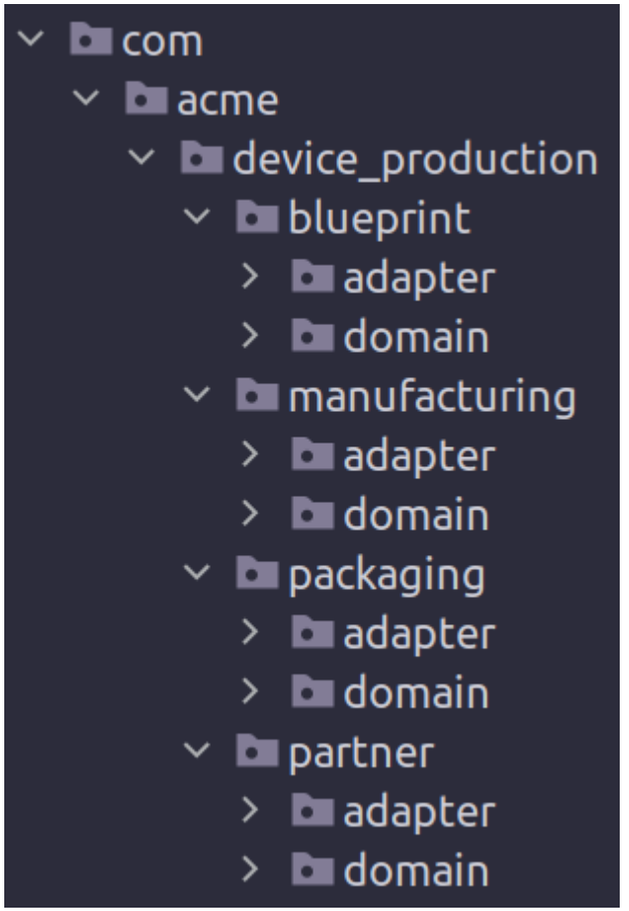
Oder via Slices-API (2)

```
@ArchTest
static final ArchRule should_respect_module_structure =
  slices().matching("com.acme.device_production.(*)..")
    .should(new ArchCondition<>("respect the module structure") {
      final Map<JavaClass, Slice> classesToModules = new HashMap<>();

      @Override
      public void init(Collection<Slice> allModules) {
        allModules.forEach(module -> module.forEach(clazz -> classesToModules.put(clazz, module)));
      }

      @Override
      public void check(Slice module, ConditionEvents events) {
        module.getDependenciesFromSelf().stream()
          .filter(it -> classesToModules.containsKey(it.getTargetClass()))
          .filter(it -> !allowedModuleDependencies.get(module.getNamePart(1))
                  .contains(classesToModules.get(it.getTargetClass()).getNamePart(1)))
          .forEach(it -> events.add(SimpleConditionEvent.violated(module, it.getDescription())));
      }
    });
```

Nun sieht unsere Struktur so aus



Problem: Kapselung?

- Package-Visibility mangelhaft
- Was ist die API?

Diskussion der API

Ergebnis: Wir definieren die API als Standard-Onion-Adapter

API-Konvention in ArchUnit

```
@ArchTest
static final ArchRule modules_should_only_depend_on_each_other_through_their_api =
  slices().matching("com.acme.device_production.(*)..")
    .should(new ArchCondition<>("only depend on each other through their API") {
      @Override
      public void check(Slice module, ConditionEvents events) {
        PackageMatcher apiPackageMatcher =
          PackageMatcher.of(".." + module.getNamePart(1) + ".adapter.module_api..");
        module.getDependenciesToSelf().stream()
          .filter(it -> !apiPackageMatcher.matches(it.getTargetClass().getPackageName()))
          .forEach(it -> events.add(SimpleConditionEvent.violated(module, it.getDescription())));
      }
    });
```

Wachstum einzelner Module

- wo nötig diskutieren und Sub-Regeln definieren
- Beispiel: Device Hierarchy

Common should be common (1)

Disclaimer: Vielleicht lieber gar kein "common" verwenden
=> Black Hole

Common should be common (2)

```
@ArchTest
static final ArchRule common_classes_should_be_used_by_multiple_modules =
  classes()
    .that().resideInAPackage("..device_production.common..")
    .and(DescribedPredicate.describe("are used by modules", clazz ->
      clazz.getDirectDependenciesToSelf().stream()
        .map(Dependency::getOriginClass)
        .anyMatch(it -> !it.getPackageName().contains("device_production.common"))))
    .should(ArchConditions.be(DescribedPredicate.describe("used by multiple modules", clazz ->
      clazz.getDirectDependenciesToSelf().stream()
        .map(Dependency::getOriginClass)
        .flatMap(BusinessStructureTest::getModuleName)
        .distinct()
        .count() > 1)));
```

Bei Verletzung

```
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
Rule 'classes that reside in a package '..device_production.common..'
and are used by modules should be used by multiple modules' was violated (1 times):
Class <c.a.d.c.a.m.StringUtils> is not used by multiple modules in (StringUtils.java:0)
```

Überblick

Konventionen müssen verstanden werden

”Ich muss noch schnell den Test grün machen”

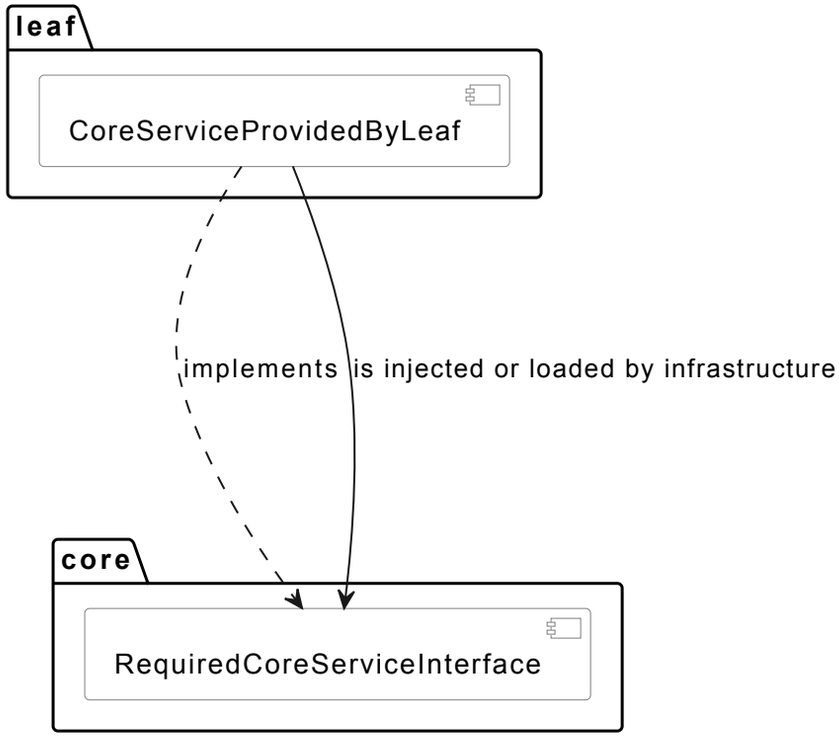
```
public String getProductSerialNumber(Object product) throws Exception {
  Class productClass = Class.forName(
    "com.acme.device_production.manufacturing.domain.model.Product");
  Method getSerialNumber = productClass.getMethod("getSerialNumber");
  return (String) getSerialNumber.invoke(product);
}
```

Diskutieren warum ein Test rot ist

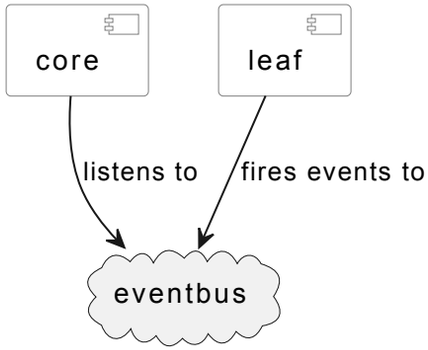
- Logik am falschen Platz?
- Fehlt ein Konzept?
- Falsche Aufteilung?
- Lösung durch Pattern?

Beispiele für Patterns

(S)ervice (P)rovider (I)nterface



Eventing



Beispiel-Problem

- Product darf nur gelöscht werden, wenn es keiner Masterbox zugeordnet ist
- Aber Module Manufacturing darf nicht auf Module Packaging zugreifen
- Mögliche Lösung: Manufacturing definiert Interface `ProductDeletionValidator` und bekommt Implementierung via Dependency Injection

Überblick

Website

https://archunit.org

Twitter

@archtests

Mastodon

@archunit@fosstodon.org

Vielen Dank für die Aufmerksamkeit :-)

Fragen?

peter.gafert@tngtech.com
https://github.com/codecholeric
@codecholeric
@codecholeric@fosstodon.org